Redressing the Balance

Burton Smith
Cray Inc.

The two types of supercomputers

- Loosely coupled systems (Type T)
  - Prices based on Transistor cost
  - Performance measured by Linpack R_{max}
  - Low bandwidth interconnection networks
  - PC-intended processors
- Tightly coupled systems (Type C)
  - Prices based on Connection cost
  - Performance measured by sparse MV multiply
  - High bandwidth interconnection networks
  - Custom processors
- Each type has its uses (and misuses)

Relative strengths

Type T:

- Arithmetic
- Well-balanced workloads
- Dense linear algebra
- Explicit methods
- Regular, non-adaptive meshes
- Slowly varying data bases
- Projects that are insensitive to programming effort

Type C:

- Data access
- Poorly balanced workloads
- Sparse linear algebra
- Implicit methods
- Irregular, adaptive meshes
- Rapidly varying data bases
- Projects that are sensitive to programming effort

Some sites have applications spanning both columns

- They may want to employ both types of system

One sort of Type C application

- A simulation problem with a wide spectrum of time and length scales may need an implicit finite difference scheme
  - Convergence may demand such an approach
  - Often the reason is merely computational feasibility
- Long-range communication will occur if the resulting time steps are big (the reason for the implicit scheme)
- Implicit methods for nonlinear PDEs solve a sparse linear system several times per time step
  - Often the sparse matrix is not very well-conditioned
- For heterogeneous problems, load imbalance is an issue
  - Irregular adaptive meshes are another source of imbalance

These attributes make climate simulation a Type C problem

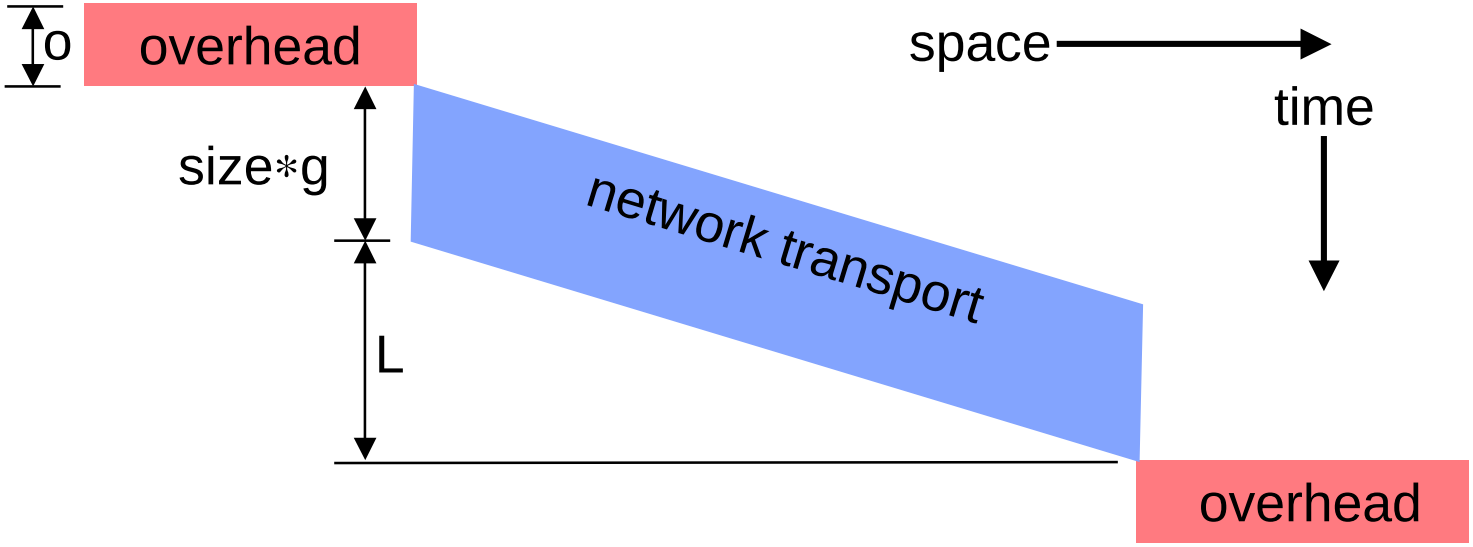
- What application did you *think* I was talking about ;-> ?

Where did all the Type C systems go?

- Type C systems have been widely deprecated
  - Some erroneously believed they cannot be scaled up
  - Others wrongly believed they are unnecessary
  - Still others keep building them, *e.g.* NEC
- The consequence was unfortunate for Type C applications
  - Most of a Type T system's resources are wasted
  - Algorithm choice is greatly restricted
  - Performance becomes highly sensitive to fine details
  - Programming becomes a heroic and frustrating task
- Type T systems can't go it alone
  - They don't have enough global bandwidth
  - This is all but inevitable given PC-intended processors
  - The reason: too much overhead

Bandwidth, overhead, and latency

- In the LogP model, well-known in computer science:
  - L is the network transport latency
  - o is the processor overhead
  - g is the reciprocal bandwidth (the “gap”)
  - P is the number of processors
- Time(size) = size∗g + 2∗o + L

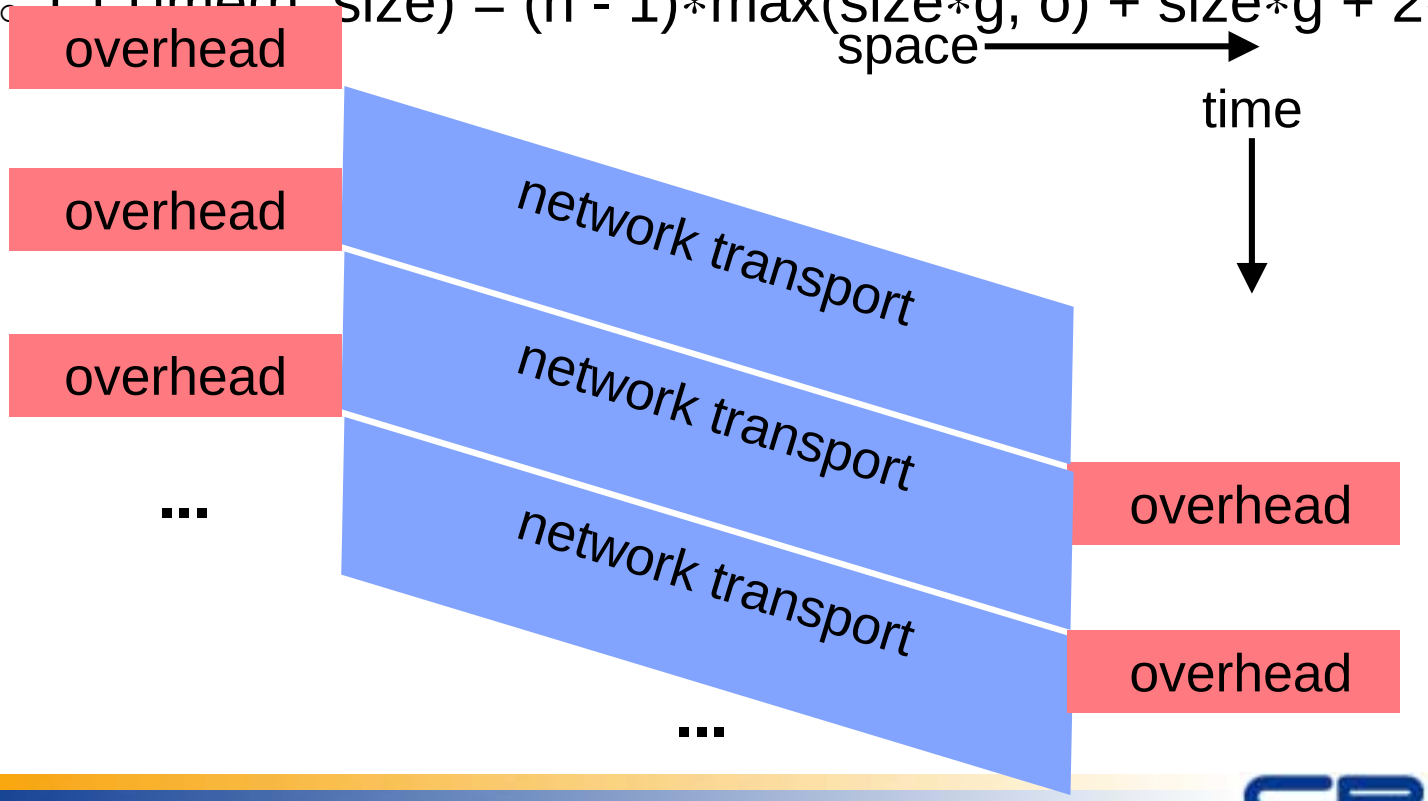


“Latency” has several meanings

- It means 2∗o + L for some, L for others
  - Each is a legitimate latency, but for different subsystems
  - Most who ask for a “low latency network” really want low o
- Some want “latency” to mean size∗g + 2∗o + L
  - This is not so useful
- We should at least try to get our names straight
  - I will use the LogP definitions

Latency tolerance (latency hiding)

- Latency can be tolerated by using parallelism
  - A new transmission can start after waiting max(size∗g, o)
  - LTTime(n, size) = (n - 1)∗max(size∗g, o) + size∗g + 2∗o



When does latency tolerance pay off?

- It depends on the relative magnitudes of size∗g, o, and L
  - n∗Time(size) = n∗(size∗g + 2∗o + L)
  - LTTime(n, size) = (n - 1)∗max(size∗g, o) + size∗g + 2∗o + L
- If size∗g >> 2∗o + L we are “bandwidth bound”
  - n-fold latency tolerance saves a mere (n - 1)∗(2∗o + L)
  - This is never significant
- If o >> size∗g + L we are “overhead bound”
  - n-fold latency tolerance saves about (n - 1)∗o
  - This will roughly halve the time
  - Unequal overheads at sender and receiver make it worse
- If L >> size∗g + 2∗o we are “latency bound”
  - n-fold latency tolerance saves approximately (n - 1)∗L
  - This is roughly an n-fold time improvement

Aside: does message size vary with P?

- Let's take PDEs as an example, and assume:
  - We have three space dimensions and one time dimension
  - We need 16 times the processors to double the resolution
- Each processor gets half as many spatial mesh points
  - If the processors are also faster, maybe somewhat more
- For nearest-neighbor communication, message size shrinks
  - Perhaps to $0.5^{2/3}$ **= 0.63 of its former size**
- For "Type C-style" all-to-all communication the message size may shrink to 1/32 of its former value
  - There are half as many mesh points per processor and 16 times as many processors to distribute data among
- Your mileage will vary, and will probably get worse

Latency tolerance in summary

- It uses parallelism to reduce total transmission time
  - It is basically just pipelined data transport
- It is most needed when size∗g is relatively small
  - either because of small size or small g (high bandwidth)
  - If size∗g is large, latency is tolerated within each message
- It is not particularly effective when overhead is high
  - This is the standard situation in Type T systems
- When both o and size∗g are small, it is invaluable
  - Vector processor references to memory
  - Multithreaded processor references to memory
- Multithreading can also tolerate synchronization latency and even branch latency
  - But that's another talk

Measuring overhead

- Ping-pong measures 2∗o + L
- Measuring latency with a logic analyzer and subtracting it from a ping-pong measurement is one idea
- Another way is to use the processor's low overhead high resolution clock to measure how long the software takes
  - What? Your microprocessor doesn't have such a clock?
- A third possibility is to vary the network latency for a pair of ping-pong measurements and do the math
  - Comparing a 10-hop route with a 20-hop, for example
  - Usually 2∗o is much greater than L so you may need a whole bunch of hops to see the difference
- A fourth way is to see how well latency tolerance works

Mitigating overhead in Type T systems

- Classic single-sided messaging doesn't help much
  - Message assembly and disassembly are expensive
- Shared memory is the *sine qua non* of low overhead
  - It will be non-uniform (NUMA), but need not be CC-NUMA
    - CC-NUMA has a scaling problem in my opinion
  - The Cray T3E and SV2 are coherent but *not* CC-NUMA
    - They tolerate latency rather than try to avoid it
- UPC, Titanium, and Co-array Fortran can help quite a bit
  - They provide a NUMA model even over fragmented memory
  - The right hardware can make the overhead pretty small
  - Since the hardware assistance is invariably sited in a coprocessor, synchronization is a source of overhead

Reducing the time-to-solution

- Type T systems are harder to program than Type C systems
  - “So many processors, so little time” - Duncan Buell, IDA-CCS
  - Or, perhaps, “the bigger they are, the harder they code”
- “Dusty deck” parallelization has several problems:
  - The high cost of whole-program analysis
  - Uncertainty about relative run-time complexities
- The MTA compilers really show what can be done
  - The level of automation is quite high, but not “dusty deck”
- Higher level programming languages are needed
  - To get beyond C++, Java, C#, and other blunt instruments
  - To make global program analysis unnecessary
- So, what ever happened to Sisal, anyway?

Improving Type T programmability

- Library-based abstractions help somewhat
  - Data layout issues are not addressed
- The aforementioned UPC, Titanium, and Co-array Fortran are higher level languages than MPI in practice
  - Since data addresses (really subscripts) are computable by the program, more sophistication is available
  - These languages work just fine on Type C systems
- ZPL kicks the level up another notch
  - Higher level operators give it “APL-like” power
  - It is pretty competitive in efficiency
  - It can also work well on Type C systems
- None of these is the silver bullet we wanted

Conclusions

- Not only are some computers unbalanced, our *field* is!
  - Most informed people seem to agree
- Redressing the balance requires Type C systems
  - Anyone arguing otherwise protests too much, methinks
- Matters are improving on this front
  - Including the possibility of some long range R&D
- And if you think architecture is “over”, Bugs Bunny, the well-known expert in persiflage, says:

  “What a maroon!”

  (I have trouble translating this in Japan and Europe)